Compiler Compilerについて (計算機構論研究会報告集)

AUTHOR(S):

萩原, 宏; 渡辺, 勝正

CITATION:

萩原, 宏 ...[et al]. Compiler Compilerについて (計算機構論研究会報告集). 数理解析研究所講究録 1969, 75: 1-26

ISSUE DATE:

1969-06

URL:

http://hdl.handle.net/2433/107973

数理解析研究所講究録
75巻 1969年 1-26

Compiler Compiler について

京都大学工学部　萩原　宏，渡辺　勝正

§1. Compiler と Compiler Compiler

すべての言語は，記号列の集合であり，Compiler（一般に translator）は，或る言語（入力言語 L_1）の記号列を，他の言語（出力言語 L_2）の記号列に変換する媒体である。

各言語の個々の記号および記号列は，ある定められた約束のもとに何らかの意味を持っており，入力言語から出力言語への変換は，系列としての記号の組み合わせだけでなく，その意味も考慮に入れておこなわれる。プログラミング言語の場合には，表現による言いまわしの違いはあっても，ニュアンス（nuance）の違いとか，あいまいさというものがないため（ないのが望ましく，あっては困る），入力記号列（source program）は一意的に出力列（object program）に変換される。

入力言語の一つの記号 a が出力言語の一つの記号 b または記号列 $b_1 b_2 \cdots b_k$ に直接対応がつく場合には，Compiler の機

構（i.e. 変換の手順）は単純である。基本的な Assembler や、一つの計算機 M_1 の機械語で書かれたプログラムを他の計算機 M_2 の機械語のプログラムに変換する手順などはその例である。

一般には、対応はもっと複雑であって、Compiler は、

(i) 構造分析過程：入力記号列 $a_1 a_2 \cdots a_m$ における各記号 a_j のむすびつきの関係を解析する。

(ii) 意味解釈過程：むすびつきの関係から明らかになった入力記号列の意味に基づいて出力列 $b_1 b_2 \cdots b_m$ を構成する。

なる二つの働をする機構を持つものである。

(1) 一つの入力言語（Problem Oriented Language, POL）と一つの出力言語（Machine Oriented Language, MOL）に対して、Compiler は一つ。

(2) 一つの POL（たとえば PL/I）と m 個の MOL に対して Compiler は m 個。

(3) p 個の POL と一つの MOL に対して Compiler は p 個。

(4) p 個の POL と m 個の MOL に対して Compiler は $p \times m$ 個。

(5) Universal Programming Language の考えに従って共通の中間言語を採用すると Compiler は $(p + m)$ 個。

(6) POL と MOL の仕様をパラメータとして与えられる場合には、一つの Compiler と $(p \times m)$ 種のパラメータ。

最後の考えに基づいて (2), (3) は次のように変更される：

(2)' 一つのPOLに対して、一つのCompilerとm種のパラメータ。

(3)' 一つのMOLに対して、一つのCompilerとp種のパラメータ。

(6)の考えに準じてCompiler Compilerを狭い意味で解釈すると：

「入出力言語L1, L2が定義された時、L1からL2へのCompilerを作り出す手順である」

と言うことが出来るが、ここでは、

「入出力言語を出来るだけ形式的(formal)に定義し、Compilerの変換手順を単純明解なものにして、計算機を用いて、Compilerをより短期間に容易に作成する手順」

を広くCompiler Compilerと解釈する。

この時、Compilerでは、入力言語L1, 出力言語L2の他にCompiler自身が表現されている記述言語L0が問題になる。すなわち、Compilerは三つの言語L1, L2, L0を規定することによって、

$$\left(\begin{matrix} L_0 \\ L_1 \longrightarrow L_2 \end{matrix}\right)$$

で特性づけられる。

更に、Compiler Compilerでは生成されるCompilerを特性づけるために、

- コンパイル方法を表わす性質 α
- Compilerの目的を表わす性質 β

がつけ加えられる。しかし、

i. コンパイル方法は最良と思われるものを採用すればよい。（入力言語 L_1 に依存する）。

ii. 新しいコンパイル方法が開発された時には、Compiler Compiler にも変更を必要とする。

iii. Compiler の目的によって、コンパイル方法も違ってくる。

などの理由のため、現在の時点では Compiler Compiler の問題を複雑にするだけであるから、当面の問題として α, β を一般的に扱うことにはしない。

§2. Compiler Compiler の方法

Compiler を特性づける三つの言語 L_0, L_1, L_2 の各々に観点をおいて、Compiler Compiler について次のような考察が加えられる。

L_0： 機械語またはアセンブラ言語で Compiler を記述する場合には、機械の特長とすぐれたプログラミング技術を活して効率の良い Compiler を作成することが出来るが、その作成とデバッグに長い期間と多くの人手を必要とする。また機械毎に Compiler を別個に記述しなければならない。これらの欠点を補う目的で Compiler 記述言語を採用することが Compiler Compiler の一方法となる。（記述型に当たる。）

L1： 入力言語を検討して出来るだけ formal な定義をふえて、単純なコンパイル方法を究明するものである。現在までになされた言語およびコンパイル方法についての理論的な研究の多くは、書きかえ規則から出発した構文論すなわち構造分析手順の解明に力が注がれてきていた。phrase structure gramnar, BNF, LR(k)-文法, canonic system 更には automaton の理論はこの部類に含まれる。その書きかえ規則の参照の仕方によってコンパイル方法は次のように分類出来る：

0. 入力言語の文法に従ってCompiler作成者がすべてを規定する。

1. 記号の順位（precedence）に基づく方法

2. Transition Matrix から分析手順をえらぶ方法

3. 書きかえ規則を直接参照しながら分析を進める方法

L2： 入力言語の意味論的考察と関連してCompilerの出力言語のあり方を検討するもので、計算機言語あるいは計算機そのものの将来の姿に一つの示唆をふえんとするものである。Universeal Language (5) の考えはその一例である。具体的には、multi-address symbolic code, λ-notation と ALGOL の対応づけ, B5500 の言語 などの研究がある。

L2 の条件としては、

{様々な入力言語から比較的容易に変換出来る。

{機械語への変換あるいは計算機による直接実行が容易。

なることが要求される。

具体的なCompiler Compilerの方法は次のように分類される。

a. 変換型 : machine m1で働くPOL pのCompiler $C^{m1}_{p\to m1}$ から machine m2で働くCompiler $C^{m2}_{p\to m2}$ を作成する。

(i) 1. $C^{P}_{p\to m2}$ を記述する

2. $C^{m1}_{p\to m1}(C^{P}_{p\to m2})$ によって $C^{m1}_{p\to m2}$ を得る。

3. $C^{m1}_{p\to m2}(C^{P}_{p\to m2})$ によって $C^{m2}_{p\to m2}$ を得る。

POL pでCompiler $C^{P}_{p\to m2}$ が記述出来るか否かが問題である。

(ii) 1. $C^{m1}_{m1\to m2}$ と $C^{m2}_{m1\to m2}$ を記述する。

2. $C^{m1}_{m1\to m2}(C^{m1}_{p\to m1})$ によって $C^{m2}_{p\to m1}$ を得る。

3. $C^{m2}_{p\to m1}$ と $C^{m2}_{m1\to m2}$ が $C^{m2}_{p\to m2}$ の役割を果たす。

m1とm2の機能・特性が似ている場合には有効である。

(iii) p1, p2 に対して $C^{m1}_{p1\to m1}$ から $C^{m2}_{p2\to m2}$ を作成する。

1. $C^{p1}_{p2\to m2}$ と $C^{p2}_{p2\to m2}$ を記述する。

2. $C^{m1}_{p1\to m1}(C^{p1}_{p2\to m2})$ によって $C^{m1}_{p2\to m2}$ を得る。

3. $C^{m1}_{p2\to m2}(C^{p2}_{p2\to m2})$ によって $C^{m2}_{p2\to m2}$ を得る。

b. Bootstrap型 : POL pのCompilerの機能を増大させてゆく。

(i) 1. pの必要十分なサブセット p_0 に対して $C^{m}_{p0\to m}$ を記述する。 $i := 0$。

2. $C^{p_i}_{p_{i+1}\to m}$ $(p_i \subset p_{i+1})$ を記述する。
3. $C^{m}_{p_i\to m}$ ($C^{p_i}_{p_{i+1}\to m}$) によって $C^{m}_{p_{i+1}\to m}$ を得る。
4. $i := i+1$ として 2, 3 をくり返えす。

k番目以降に対しては $p_k, p_{k+1}, \cdots$ のうち Compiler の記述に必要な部分は同じになると考えられる。その様な k をみつけることが、Compiler記述言語に関する一つの問題である。

(ii) 1. $C^{m}_{p_0\to m}$ を記述し、$i := 0$。
2. $C^{p_i}_{p_{i+1}\to p_i}$ を記述する
3. $C^{m}_{p_i\to m}$ ($C^{p_i}_{p_{i+1}\to p_i}$) によって $C^{m}_{p_{i+1}\to p_i}$ を得る。
4. $\overset{i}{\underset{j=i}{\cup}} C^{m}_{p_{j+1}\to p_j}$ と $C^{m}_{p_0\to m}$ が $C^{m}_{p_{i+1}\to m}$ の役割をする。
5. $i := i+1$ に対して 2, 3, 4 をくり返えす。

作成された Compiler は multi phase になる。

C. 記述型 : Compiler 作成の手数を軽減し、計算機に独立に Compile 過程を記述するために、記述言語 L_0 を定義する。

(i) macro assembler 言語を L_0 に用いる。Compiler の基本となる動作を macro order として定義しておく。

(ii) ALGOL や PL/I などの POL を L_0 とする。a および b の手法が適用出来るが、ALGOL では付加すべき機能が問題となり、PL/I などはサブセット p_0 の選択が問題になる。また、構造分析手順の記述が繁雑になる可能性が大きい。

(iii) 記号処理を行う List Processing 言語を L_0 とする。

(iv) "Compiler を記述する" という問題向き言語と定義する。

Brooker R.A : The Compiler Compiler.
Annual Review in Automatic Programming Vol 3. 1963
Feldman J.A : Translator Writing System.
C. ACM Vol 11 (Feb. 1968)

定義された Lo によって Compiler を簡潔に表現出来るが、次の二つが短所となる。

1. Lo として採用する言語によって、コンパイル方法がある程度制限される。
2. 新たな言語 Lo の Processor を必要とする。そのため Lo の条件としては、理解しやすく書きやすいこと、Compiler の動作が完全に記述出来ること、によって、Lo の Processor が出来るだけ簡単であること、が要求される。

d. Parameter型 : §1 の (6) または (3)' に基づいて POL (および MOL) の定義を何らかの形で与えると、あらかじめ用意された構造分析手順に従ってコンパイルを行う Compiler が構成される。Syntax-directed Compiler はその典型である。

d. と c (iv) とは本質的に区別を明確にしがたいが、

構造分析手順	は	d	Ex. Production Language
意味解釈手順	は	c (iv)	Formal Semantic Language

の傾向が強い。

様々の Compiler Compiler の方式が考えられ、いくつかの試みがなされているが、その実用性と作成された Compiler の space と compile-time に問題が残されている。

次に、記述型方式の一例としての Compiler 記述言語 COL とその根底となる syntax directed analysis について検討する。

COL は、

- 能率のよい Compiler を、より容易に作成する。
- Compiler が何を為すものであるかを明らかにする。
- Compiler にとって必要不可欠な機能を知ることによって、コンパイルを行う計算機あるいは入力言語を直接理解して実行する計算機にそなわるべき機能を探究する。

ためのものである。

§3. Syntax-directed Analysis の問題点.

phrase structure grammar $G = (V, \Sigma, P, \sigma)$ において、書きかえ規則の組 P は、

$$A \longrightarrow X_1 X_2 \cdots X_m \quad , \quad A \in N, \ X_i \in V$$

でよられるものとする。ここに、

V : vocabulary

Σ : alphabet i.e. a set of terminal symbols $\Sigma \subset V$

$N = V - \Sigma$, $N \cap \Sigma = \phi$

P : a set of rewriting rules

σ : initial symbol $\quad \sigma \in N$

phrase structure grammar G によって言語 $L(G)$ が定義される。

$$L(G) = \{ \alpha \mid \alpha \in \Sigma^*, \sigma \overset{*}{\Rightarrow} \alpha \}.$$

3.1 Top down 分析法

入力記号列を tree 構造に表現したとき，tree の root にある node σ からはじめて、各 node を構成する要素を、与えられた書きかえ規則の組 P を参照しながら、順に確認し，最終的に入力記号列 $\alpha = a_1 a_2 \cdots a_n$ が σ に帰着されることを確かめるものである。これは sequential machine の状態図に基づいて状態を移行させてゆくことによく似ている。

規則の組 P は次の諸点に注意して与えられる：

a. 同じ左辺の記号を持つ規則をひとまとめにして、node σ を持つ規則から順にならべる。この時，規則の順番が参照順序になるため

$$A \longrightarrow \alpha_1$$

$$A \longrightarrow \alpha_2$$

なる二つの規則に対して、α_2 が α_1 の左の部分記号列となる可能性のある場合には、後者を後順にする。

例1. $\langle$for list element$\rangle \longrightarrow \langle$ae$\rangle$ <u>step</u> $\langle$ae$\rangle$ <u>until</u> $\langle$ae$\rangle$

$\langle$for list element$\rangle \longrightarrow \langle$ae$\rangle$

b．左回帰的規則（left recursive rule），たとえば

$$A \longrightarrow Ab$$

は，回帰的定義の出発となる規則

$$A \longrightarrow a$$

と合わせてくり返しの形に表現する。すなわち，

$$A \longrightarrow Ab \Longrightarrow Abb \Longrightarrow \cdots \Longrightarrow abb\cdots b = a\,b^{*}$$

を

$$A \longrightarrow a\,*\{\,b\,\}$$

と表わす。$*\{\ \}$は$\{\ \}$の中の要素が零回または任意回くり返して規則の構成要素となっていることを示す。

例2．
$$\begin{cases} \langle \text{block head} \rangle \longrightarrow \underline{\text{begin}}\ \langle \text{declaration} \rangle \\ \langle \text{block head} \rangle \longrightarrow \langle \text{block head} \rangle\ ;\ \langle \text{declaration} \rangle \end{cases}$$

は

$$\langle \text{block head} \rangle \longrightarrow \underline{\text{begin}}\ \langle \text{declaration} \rangle\ *\{\ ;\ \langle \text{declaration} \rangle\}$$

と表示される。

c．右回帰的規則（right recursive rule）もくり返しの形に表示する。

例3．
$$\begin{cases} \langle \text{compound tail} \rangle \longrightarrow \langle \text{statement} \rangle\ \underline{\text{end}} \\ \langle \text{compound tail} \rangle \longrightarrow \langle \text{statement} \rangle\ ;\ \langle \text{compound tail} \rangle \\ \langle \text{compound tail} \rangle \longrightarrow \langle \text{statement} \rangle\ *\{\ ;\ \langle \text{statement} \rangle\}\ \underline{\text{end}} \end{cases}$$

d．条件aに従う範囲内で，同じ左辺の記号を持つ規則は参照頻度の高い順にならべる。これは分析速度をあげるためのもので手順の本質にはかかわりない。

例4. <basic statement> ⟶ <assignment statement>

<basic statement> ⟶ <go to statement>

<basic statement> ⟶ <procedure statement>

<basic statement> ⟶ <dummy statement>

e. 同じ左辺の記号を持つ規則が右辺が共通の部分記号列を持つ場合には、次のいずれかによって規則を変形する。

e1. 新たな nonterminal symbol を導入する

e2. factoring とよぶ形に変形する。

例5. <term> は次のように定義される。

e1: <term> ⟶ <factor> * { <post factor> }

<post factor> ⟶ × <factor>

<post factor> ⟶ / <factor>

<post factor> ⟶ ÷ <factor>

e2: <term> ⟶ <factor> * { × <factor>

| / <factor> | ÷ <factor> }

以上の変形の後に、第p番目の規則は、

p: $A_p \longrightarrow X_{p1} X_{p2} \cdots X_{pnp}$ [w_p]

でよられる。ここに、w_pは規則pが成立した時に出力列に加えるべき記号列あるいは規則の番号pを表わす。くり返し要素に対しては、分析の結果、何回適用されたかが明示される必要がある。(手順の詳細は省略する)

3.2 Bottom up 分析法

node σ を最終目標として、入力記号列 $\alpha = a_1 a_2 \cdots a_m$ が σ に帰着されることを確認するわけであるが、Bottom up では、ともかく入力記号 a_1 を読んで、当面の目標（σ）に関係なく、a_1 をオー構成要素に持つ規則 g

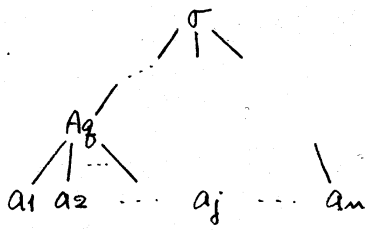
$$g : A_g \longrightarrow a_1 X_{g2} \cdots X_{gng}$$

を参照して、この規則の成立を確認する。規則 g が成立したとき、$A_g = \sigma$ なら分析は完了するが、$A_g \neq \sigma$ の場合には A_g をオー構成要素に持つ規則 g'

$$g' : A_{g'} \longrightarrow A_g X_{g'2} \cdots X_{g'mg'}$$

を参照する。

参照する規則が、次の入力記号あるいは確認された node（これらは目標とする node の最も左の記号である）によって選択されるため、書きかえ規則の左辺と右辺を入れかえて、



$$g : X_{g1} X_{g2} \cdots X_{gng} \longrightarrow A_g \quad [w_g]$$

と表現し、オー構成要素に同じものを持つ規則をまとめて順序づける。この時、条件 a と e には注意する必要があるが、規則の回帰性（条件 b，c）には特に配慮する必要はない。回帰的規則は何回かくり返して参照される。

例6. <assignment statement> の構造分析をする規則の組はそれぞれ次のように与えられる。

$P_{TOP\ DOWN}$

1 <as> ⟶ <left> <e>
2 <e> ⟶ <t> *{<ap> <t>}
3 <t> ⟶ <p> *{<mp> <p>}
4 <p> ⟶ I
5 <p> ⟶ (<e>)
6 <left> ⟶ I :=
7 <ap> ⟶ +
8 <ap> ⟶ −
9 <mp> ⟶ ×
10 <mp> ⟶ /

$P_{BOTTOM\ UP}$

1 <left> <e> ⟶ <as>
2 <e> <ap> <t> ⟶ <e>
3 <t> <mp> <p> ⟶ <t>
4 <t> ⟶ <e>
5 <p> ⟶ <t>
6 I := ⟶ <left>
7 I ⟶ <p>
8 (<e>) ⟶ <p>
9 + ⟶ <ap>
10 − ⟶ <ap>
11 × ⟶ <mp>
12 / ⟶ <mp>

分析過程を能率良く進めるために、規則の組Pから次の性質を調べて一覧にしておく。

I. 規則 q の構成要素 X_{qi} ($1 \leq i \leq m_q$) は、目標 A_q に対して一種の到達可能性を持つと言う。

II. 規則 q の右一要素 X_{q1} に対して、A_q が規則 r の右一要素であるか、A_q が X_{r1} に対して一種の到達可能性を持つ場合、

Xq_1 は Ar に n 種の到達可能性を持つと言う。

出発 ＼ 目標	<as>	<left>	<e>	<t>	<p>	<ap>	<mp>
<left>	1						
<e>	1		1		1		
<t>			1	1			
<p>			2	1			
<ap>			1				
<mp>				1			
I	2	1	2	2	1		
:=		1					
(			2	2	1		
+ −						1	
× /							1
)					1		

定義. 記号 X ($X \in V$) から目標 A ($A \in N$) に n 種または n 種の到達可能性を持つ時、Ar に到達可能といい、

$$T(X, A) = 1$$

と表わす。そうでない時は

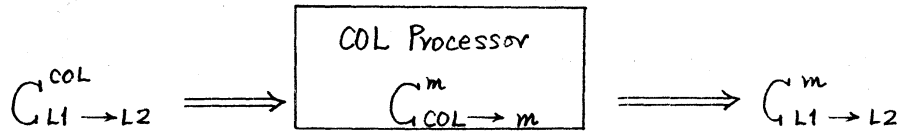
$$T(X, A) = 0.$$

例. $T(<p>, <e>) = 1$, $T(<p>, <t>) = 1$

$T(<p>, <p>) = 0$, $T(<p>, <as>) = 0$

§4. Compiler 記述言語：COL

COL システムは、計算機によって Compiler を作り出すプログラミング方式の一つで、COL 言語と各計算機に対する COL-Processor の二つの要素に分けられる。



COL 言語は Compiler の構造分析過程と意味解釈過程を記述する能力を持ち、前者を Syntax Statement, 後者を Semantic Statement によって表現する。生成される Compiler は、

<COMPILER> ::= *{ Syntax Statement } ENDMARK
*{ <M-routine> } ENDMARK

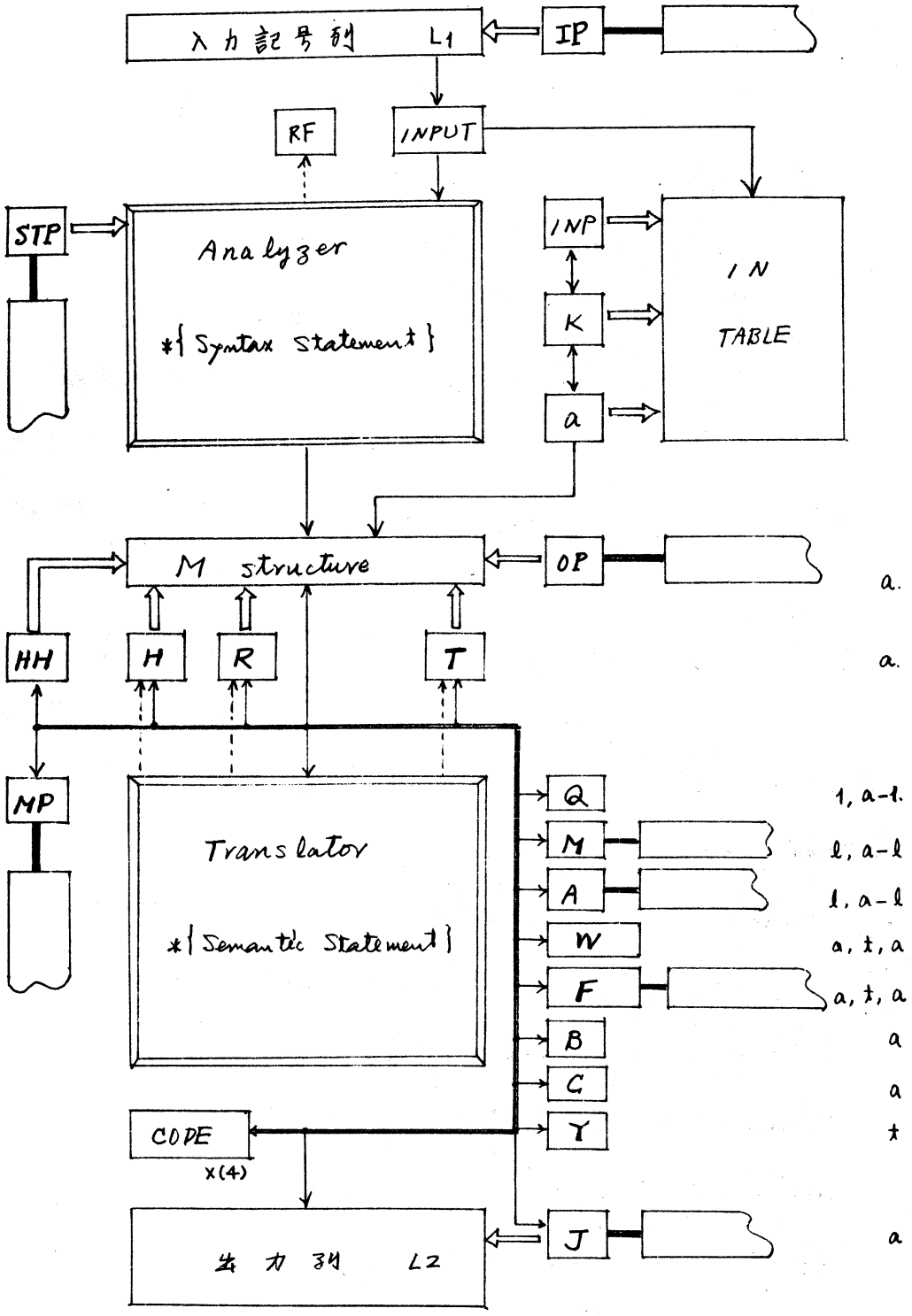
<M-routine> ::= MNAME : *{ Semantic Statement }

で定義され、次1回の構成を持つ。

1. Syntax Statement の記述から変換される部分は、入力言語の記号列の構造を分析して、結果を表わす M-structure を構成する。[Analyzer].
2. Semantic Statement の記述から変換される部分は、Analyzer の解析結果に基づいて意味解釈を行ない、目的とする出力列を構成する。[Translator].

4.1 Syntax Statement

Syntax-directed analysis における Compiler の動作を分類する


入力記号列 L1
IP
RF
INPUT
STP
Analyzer
*{ Syntax Statement }
INP
K
a
IN
TABLE
M structure
OP
a.
HH
H
R
T
a.
MP
Translator
*{ Semantic Statement }
Q
1, a-1.
M
l, a-l
A
l, a-l
W
a, t, a
F
a, t, a
B
a
C
a
Y
t
CODE
x(4)
出力列 L2
J
a

第1図

と次のようになる。

i　terminal symbol となる "delimiter" の確認.

ii　"identifier" "number" "string" などの basic item の編集と変換.

iii　nonterminal symbol となる syntactic unit の確認とその応答.

iv　iii の手順に必要な情報の記憶と復元.

v　分析手順の次の動作の指定 i.e. 参照規則の選択

vi　分析結果の記録

COL による構造分析過程は規則の組 P と分析手順が一つにまとめられて、一連の Syntax statement で記述される。

< Syntax statement > ::= < Label > < Read & Test > ?
< T. Action > & < F. Action >

< Label > ::= SNAME : | Φ

< Read & Test > ::= < Read > < Test >

< Read > ::= ⁙ | < Read > ⁙ | Φ

< Test > ::= I | N | ¢ | "delimiter" | (SNAME)

< T. Action > ::= < Trans > | *{ < Action > , } < Trans >

< Trans > ::= TR | FR | # SNAME | Φ

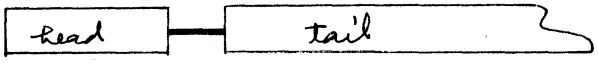
< Action > ::= < Result > | ON | / | < Error >

< Result > ::= (MNAME) | @

< Error > ::= E* { letter | digit }

< F. Action > ::= < T. Action > | < T. Action > , AT

push down 記憶は



の構造を持ち、次の指令によって操作される。

RESERVE (E) : tail E push down ; head E → tail E ;

RESTORE (E) : head E ← tail E ; tail E pop up ;

REWRITE (E) : head E ← tail E ;

LOSE (E) : tail E pop up ;

ERASE (E) : tail E をすべて消去する ;

4．2 M structure

入力記号列の構造分析の結果、MNAME (Mroutine の名前) と basic item の内部表現から成る Mstructure が構成される。MNAME は入力記号列の文脈を表示すると共に、意味解釈過程で実行される Mroutine を指示している。すなわち、Mstructure は入力記号列を標準型に表現したものであって、意味解釈を容易にして、その手順を簡潔にするための媒介となる。

Mstructure は、入力記号列の文脈が確定する Incremental Unit (IU) 毎に構成されて、意味解釈過程におくられる。

4．3 Semantic Statement

COL では Semantic Statement によって意味解釈過程すなわち入力言語 (たとえば ALGOL) の意味が表現される。これは第1図の register および記憶装置を持つ機械を作動させるも

のである。

<Semantic Statement> ::= <Uncond st> ; | <Cond st> ;
| MNAME : <Semantic Statement>

<Uncond st> ::= <Asst> | <Objest> | <Mtrans> | <Point> |
<Pdstack> | <Link> | <Error>

<Asst> ::= <Left> = <Right>

<Cond st> ::= & <Eqitem> = <Eqitem> ? <Uncond st>

<Objest> ::= $ <Op> * { , <Od> }

<Mtrans> ::= # MNAME | ¢ MNAME | /

<Point> ::= (+ VP) | (- VP)

<Pdstack> ::= { RESERVE | RESTORE | REWRITE | LOSE | ERASE } (Vpd) |
SEARCH (V) | DELETE (V) | FIND

<Link> ::= LINK (V, V)

<Error> ::= E * { letter | digit }

4. 4 Compile-Time

FACOM 230-10 FORTRAN	FACOM 230-10 ALGOL	Compiler I	Compiler II	Compiler III	Compiler N
1	0.8 ~ 0.9	1.5 ~ 2	2 ~ 3	4 ~ 6	3 ~ 4

Brooker : { space ~ handcoded version の 1.5 ~ 2 倍
compile time ~ 4 から 5 倍

Brooker R.A : Experience with the Compiler Compiler
Comp. Journal Vol 9 (1967)

時間のかかる理由

1. 構造分析と意味解釈をIU毎に行っているため.

状態の変化
必要ページの取り出し } の回数が多くなる。

2. Syntax-directed analysis のため

push down の操作が多い (特に aritmetic expression)
入力記号の読み直しが多い

§5. 様々の問題

COLによってCompilerを記述・作成した結果，遭遇しないくつかの問題点を、一般のCompiler Compilerの場合を含めて検討する。

5.1 受け入れられる入力言語L1とコンパイル方法

Compiler記述言語がすべてのアルゴリズムを記述表現出来る場合には問題はないが，一般にコンパイル手順を簡潔にするためにCompiler記述言語を定義した場合には、コンパイル方法が何らかの形で規定され、それによって受け入れられる(accepted)入力言語L1のクラスが制限されると考えられる。

a. COLで記述されたCompiler $C^{COL}_{L1\to L2}$では

i. 構造分析手順はLR(k)文法に従う言語を受け入れる。

ii. $C^{COL}_{L1\to L2}$の構造分析過程はpushdown automaton，意味解釈

過程は stack automaton にモデル化して考えられるが、

"L を任意の CS-言語とした時、$T(A) = L$ となる stack automaton A が存在する。"

"L を任意の CF-言語とした時、$T(M) = L$ となる push down automaton M が存在する。"

によって、$C^{COL}_{L1 \to L1}$ は CF-言語を受け入れることが出来る。

b. 構造分析手順は Syntax-directed analysis に準じているため arithmetic expression のように対応の tree 構造の node が深くなるものは構造分析に時間がかかる。そのためコンパイル方法を operator precedence に基づいた方法などに変更したいが、COL そのままでは少し困難である。

c. IU の選び方によって Semantic Routine (M-routine) の記述が変ってくる。たとえば、<actual parameter> を IU にえらぶと <expression> の途中で M structure が切れる可能性があるため、<assignment statement> に関する M routine が少し変更される。

d. 記憶の割りはり方法 (static or dynamic)、side effect を考慮するか否か、その他入力言語の意味論的性質によって必要な記述能力が変ってくる。

e. 添字付変数の番地計算と FOR ループ (DO ループ) の関係のように、コンパイル結果の optimization を行う場合、そ

の手順が記述出来るか否かが問題となる。

<d.e. は特に Parameter型の場合にはむつかしくなる。>

5．2 計算機の機能・特徴について

コンパイルを行う計算機（C-time machine）およびコンパイル結果の出力列に従って実行する計算機（R-time machine）の特徴に COL Processor および出力言語L2が大きく依存する。

a．R-time machine の register 構成に合った出力列が生成されるか否か。（Cf. Compiler I と Compiler N）

b．C-time machine の入力機能に大きく依存する Read Operation および "delimiter" operation は、あらかじめ準備された C-time Subroutine におきかえられる。

c．array の記憶番地の割付けと計算
　procedure の呼び出し手順
　入出力 format の処理
　などの R-time Subroutine は、一般に R-time machine 毎に準備される。

d．計算機の記憶装置の状態に応じて Compiler および出力列が構成される。FACOM 230-10 ではページ構成になる。

—— Compiler 作成の問題においても、人間－機械系における人間と計算機の仕事の分担をどのようにするかが、一番大きな問題となる。 ——

Compiler I

◦入力言語

<BLOCK> ::= begin *{<D>} <ST> *{; <ST>} end &

<D> ::= real I *{, I} ;

<ST> ::= I = *{I =} <EXP>

<EXP> ::= {− | φ} <T> *{+ <T> | − <T>}

<T> ::= <F> *{× <F>}

<F> ::= <P> *{! <P>}

<P> ::= I | N | (<EXP>)

ただし I, N はそれぞれ identifier, number を表わす。

◦M-structure

<p> = [P] a | [N] a | <ae>

<f> = <p> *{<p> [EXP]}

<t> = <f> *{<f> [MUL]}

<ae> = <t>{[NEG] | φ} *{<t> [ADD] | <t> [SUB]}

<st> = [P] a [LEFT] *{[P] a [LEFT]} <ae> [RITH] /

<d> = [REAL] [I] a *{[I] a} [TD] /

<block> = [HEAD] *{<d> [DH]}

<st> *{[STE] / <st>} [END] /

ただし、/ は IU の切れ目を表わし

[ ]中の名前は MNAME である。

```
° SYNTAX TABLE

BL:     ※"°BEGIN°"     ? (HEAD)          & #BL
BD:     (D)             ? (DH),#BD        &
BS:     (ST)            ?                 &
        ※";"            ? (STE)/ #BS      &
        "°END°"         ? (END)/          & E100,AT
EE:     ※"&"            ? TR              & #EE
D:      ※"°REAL°"       ? (REAL)          & FR
D1:     I               ? (I) @           & E101
        ※","            ? #D1             &
        ";"             ? (TD)/ TR        & E102,AT
ST:     (LEFT)          ? #ST             &
        (EXP)           ? (RITH)/TR       & (DUMY) TR
LEFT:   I               ? (P)@            & FR
        "="             ? (LEFT) TR       & FR
EXP:    ※"-"            ?                 & ON,#EP1
        (T)             ? (NEG) #EP2      & (NUL)#EP2
EP1:    (T)             ?                 & FR
EP2:    ※"+"            ?                 & #EP3
        (T)             ? (ADD)#EP2       & E201,(NUL) AT
EP3:    "-"             ?                 & ON,TR
        (T)             ? (SUB)#EP2       & E202 (NUL) AT
T:      (F)             ?                 & FR
T1:     ※"※"            ?                 & ON,TR
        (F)             ? (MUL) #T1       & E203 (ONE),AT
F:      (P)             ?                 & FR
F1:     ※"!"            ?                 & ON,TR
        (P)             ? (EXP) #F1       & E204 (ONE) AT
P:      I               ? (P)@,TR         &
        N               ? (N) @, TR       &
        ※"("            ?                 & FR
        (EXP)           ?                 &
        ※")"            ? TR              & E205,ON,TR

SYNTAX END
```

```
o M-ROUTINES

HEAD:     ERASE(M, F, A, J); M1=1; M2=0;
          J=0800; A1=1; A2=1;
          F1=8100; F2=0; F3=0; B=0;
          RESERVE(M,A,F,J);
                                $"STAT" M; /;
END:      LOSE( M );  RESTORE( M );
          RESTORE(J);           $"HALT" J; /;
STE:      REWRITE( M ) /;
REAL:     Y=1/;
I:        (+R); F1=(R); F2=Y;
          (+M2); F3=M; RESERVE(F); (+B)/;
TD:       Y=0/;
DH:       &(T)=(TD)? /;
          RESERVE(M);/;
P:        (+R); F1=(R);
          FIND; F=F(02); (+H); (H)=F3; (+H); (H)=F2;/;
LEFT:     (T)=(AG); (+T);
          (T)=(H); (+T,-H); (T)=(H); (+T,-H);
          (T)=(RITH)/
AG:       (+R);(+R);(-H);     $"STOR" (H),(R);
          (+H)/;
RITH:     /;
DUMY:     /;
NEG:      (-H); ¢TMU;           $"FNEG" (H),Q;
          (H)=0; (+H)/;
ADD:      CODE=" ADD";   #OP;
SUB:      CODE=" SUB";   #OP;
MUL:      CODE=" MUL";   #OP;
EXP:      CODE=" EXP";
OP:       CODE=CODE+"F    ";
          (-H); ¢TMB;           $CODE, (H-4),(H), 0;
          (-H); (H-2)=0/;
TMB:      Q=(H-4); &01=1? #CHEK;
TMU:      Q=(H);    &01=1? #MSET; #TEMP;
CHEK:     02=02-1;
          & Q=(H) ? #TMU;
          02=(H-4);
MSET:     M=02;/;
TEMP:     (+M2); 02=M; Q1=1 /;
N:        (+R); Q=(R);
          (+H); (H)= 02; (+H); (H)=Q1+1;/;
NUL:      (+H); (H)=1X00; (+H); (H)=1/;
ONE:      (+H); (H)=1X01; (+H); Y.7=1; (H)=Y/;

M ROUTIN END
```